Emerging Technologies for V&V of ISHM Software for Space Exploration

Martin S. Feather
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr
Pasadena, CA 91109
818-354-1194
Martin.S.Feather@jpl.nasa.gov

Lawrence Z. Markosian
QSS Group, Inc.
NASA Ames Research Center
Moffett Field, CA 94035
650-604-6207
lzmarkosian@email.arc.nasa.gov

Abstract—Systems[1,2] required to exhibit high operational reliability often rely on some form of fault protection to recognize and respond to faults, preventing faults' escalation to catastrophic failures. Integrated System Health Management (ISHM) extends the functionality of fault protection to both scale to more complex systems (and systems of systems), and to maintain capability rather than just avert catastrophe. Forms of ISHM have been utilized to good effect in the maintenance phase of systems' total lifecycles (often referred to as "condition-based maintenance"), but less so in a "fault protection" role during actual operations. One of the impediments to such use lies in the challenges of verification, validation and certification of ISHM systems themselves. This paper makes the case that state-of-the-practice V&V and certification techniques will not suffice for emerging forms of ISHM systems; however, a number of maturing software engineering assurance technologies show particular promise for addressing these ISHM V&V challenges.

TABLE OF CONTENTS

1. INTRODUCTION

There is increasing interest in Integrated System Health Management (ISHM) for space vehicles, including human-rated space transportation systems [1], [2]. ISHM goes beyond fault protection (FP) and fault detection, isolation and recovery (FDIR) by monitoring and predicting system performance, diagnosing faults, and planning and even controlling vehicle behavior in the presence of faults and other off-nominal scenarios. Future spacecraft will operate autonomously while orbiting the moon and planets for extended periods of time while their entire crew descends to the surface in a separate lander. Crew members and ground controllers will be required to communicate with the orbiting spacecraft and monitor its "vital signs" remotely. By way of contrast, during the Apollo era, one astronaut stayed with the "mother ship", while the lunar lander carrying two astronauts descended to the moon.

The human-rating requirements for US spacecraft includes a "two fault-tolerance" requirement that it shall be able to detect, isolate and recover from two subsystem failures. By comparison, Apollo generally had only single fault tolerance.

A major impediment to ISHM acceptance is the perceived inability to certify it for mission- and safety-critical systems: it is necessary to show that ISHM, in concert with the system it manages, indeed exhibits the required levels of reliability, and that ISHM cost-effectively increases the reliability of the entire system over the reliability of the underlying system without ISHM. Unlike other possible contexts for ISHM, in the spacecraft context it is infeasible to implement it and gather the evidence demonstrating its success over a long period of representative testing.

The principal thesis of this paper is that ISHM faces significant V&V impediments: specifically, current state of the practice V&V and certification techniques do not scale up to the challenges that ISHM poses; however, there are emerging V&V technologies that address these challenges.

In the more general context of critical software, the challenges posed by FAA-mandated certification of aerospace software were outlined several years ago [3]. A similar perspective on this same issue is reported in [4]. V&V issues for advanced software technologies planned for a spacecraft's use were the topic of [5].

The next section summarizes ISHM software challenges. Then we discuss a variety of emerging V&V/certification technologies that are potentially enabling for ISHM and

1

[1] 0-7803-9546-8/06/$20.00© 2006 IEEE

[2] IEEE Aerospace Conference paper #1441, V-2, Nov 11, 2005

ISHM-like systems. Section 3 describes development technologies that complement, and in some cases enable, the V&V technologies. Those sections provide evidence that current planning for ISHM should include planning for the maturation and application of these V&V technologies. Section 4 then discusses the interplay between development practices and V&V/certification challenges. Section 5 provides our conclusions. Finally we provide an extensive bibliography documenting relevant research directions.

2. ISHM Software Challenges

There are several fundamental reasons that V&V of ISHM is difficult. First, most of the scenarios that ISHM is designed to manage are *off-nominal*. For these scenarios, it is hard to know that all the significant possible failure modes have been identified; and for any failure mode, its characteristics may not be well understood.

Also, there is a large number of off-nominal scenarios. For example, if there are 1,000 possible failures, then there are potentially 1,000,000 *pairs* of such failures. Even if not all of these combinations are possible, the order-of-magnitude number of pairwise combinations grows as the square of the number of individual possible failures.

Thus, ISHM systems are challenging in terms of the sheer number of their possible executions—they exhibit a large "state space". Human rating requirements state that testing is required to "verify and validate the performance, security, and reliability of all critical software across the entire performance envelope", which includes much of this state space.

In addition, ISHM must not only detect and handle off-nominal scenarios, but it must avoid "false alarms". For example, ISHM needs to distinguish engine failure from failure of the sensors monitoring the engine's health; the algorithms that perform this function must be extremely reliable, since they are in continuous operation.

For a survey of the state of the practice for V&V of ISHM for space exploration, see [6], which discusses both the limitations on current ISHM V&V and the inability of current practice to scale up.

The preceding challenges to V&V of ISHM are derived from the requirements levied on ISHM. In addition, there are challenges that arise from the way ISHM is designed and implemented. We discuss the second set of challenges in the next section, since the appropriate V&V technologies are often specific to design and implementation techniques.

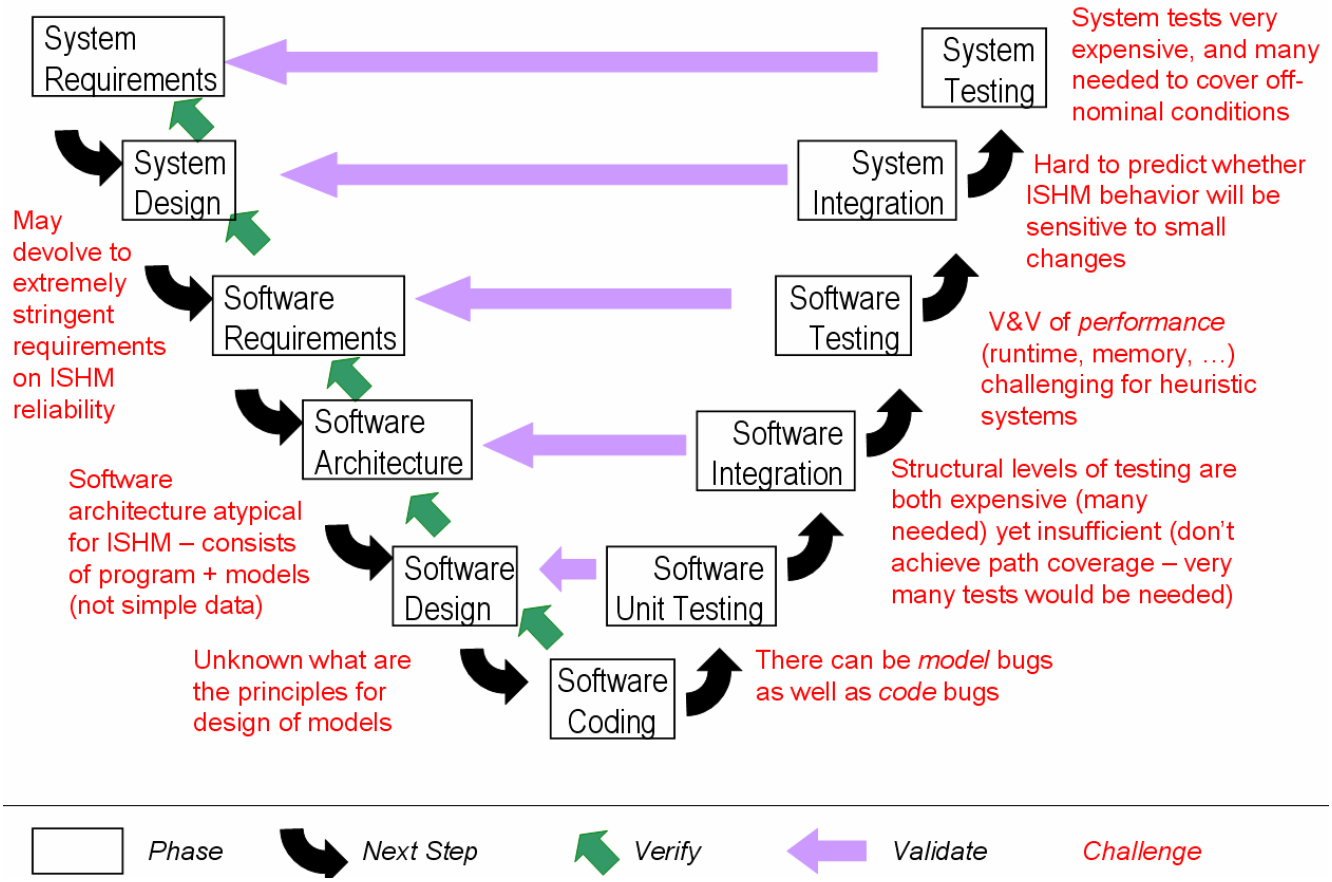


Figure 1: ISHM Lifecycle Stages and V&V Challenges

3. EMERGING V&V TECHNOLOGIES

The unusual role and nature of ISHM software raises both challenges for V&V and certification, and opportunities to amplify the efficacy of existing techniques, and to make use of some new and emerging V&V techniques that offer the promise of overcoming some of those key challenges. This section describes the origins of those opportunities, and gives some representative examples of emerging V&V techniques.

ISHM Architecture

Emerging forms of ISHM are likely to be architected using a combination of hierarchical composition and model-based reasoning. In hierarchical composition, each subsystem will perform its own health management, and will propagate its status, and if necessary the faults it cannot manage locally, to the system of which it is a part. In model-based reasoning, a generic reasoning engine will operate over system-specific models.

Hierarchical composition potentially favors V&V by allowing analysis itself to take advantage of the hierarchy, subdividing the V&V into manageable portions. V&V of this kind, often referred to as "hierarchical verification" or "compositional verification", is an area of current interest within the V&V community. For a discussion of some of the issues, see [7]; for an example of a whole workshop focused on the topic, see [8]. Some of this work has been applied to NASA missions, e.g., [9]. Compositional verification, while potentially usable with a variety of verification techniques, often incorporates model checking (discussed separately below) to verify system designs, but the combination can also be applied to verify the source code as well. [10] discusses an application to a NASA robotic controller.

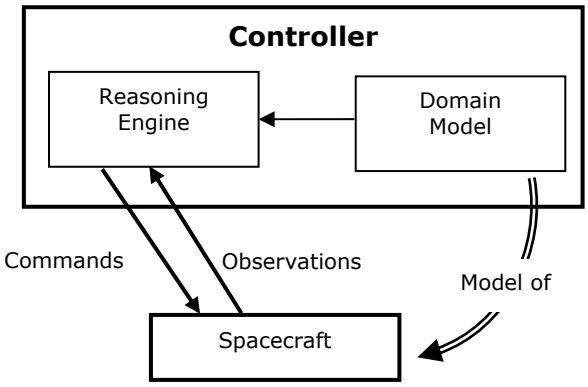


Figure 2: Model-Based ISHM Architecture

Model-based approaches to ISHM yield an ISHM system architecture divided into a generic, and therefore reusable, reasoning engine, and system-specific models as shown in Figure 2 (adapted from [11]). The reasoning engine itself is a non-trivial piece of software, and so the correctness of its implementation needs to be checked. However, since it will be reused from application to application, the effort it takes to check that implementation can be amortized over those multiple applications. An example of this is in [12], where validation is performed on properties of the core of a spacecraft's fault protection system—the core "engine" orchestrates the handling of fault reports and the running of responses to handle them.

Models Used in ISHM

In order that ISHM can perform its reasoning (e.g., diagnose the cause of a fault from a set of symptoms), those models are designed to be machine-manipulable by the ISHM reasoning engine itself. V&V can also benefit from such machine-manipulable models. As stated in [13], "These models are often declarative and V&V analysts can exploit such declarative knowledge for their analysis".

Many of the emerging V&V techniques perform analysis – for V&V purposes – over the same kinds of models that ISHM utilizes. The adoption of those V&V techniques in *traditional* software settings has always been impeded by the need to construct such models by hand, from the various forms of system documentation intended for human, but not computer, perusal (e.g., requirements stated in paragraphs of English). Hand-construction of V&V models is expensive and time-consuming, and as a result their application has, in practice, been limited to only the most critical core elements of software and system designs (for an in-depth discussion, see [14]). A representative example drawn from the spacecraft fault protection domain is [15]'s use of "model checking" applied to the checkpoint and rollback scheme of a dually redundant spacecraft controller. In contrast, in model-based ISHM, such models are available early in the lifecycle, the ideal time to benefit from the results of analysis. Automatic translation from the form of ISHM-like models to the form of V&V models has been shown to be feasible, e.g., [16] illustrates such an approach in which they translate statecharts into the input form for the model checker SPIN (see Figure 3 below); [17] translate models in Livingstone (a model-based health management system [18]) into the model checker SMV [19] reports experiments to translate AI planner domain models into SMV, SPIN and Murphi model checkers, allowing a comparison of how the different systems would support specific types of validation tasks. We discuss model checking in more detail later in this section.

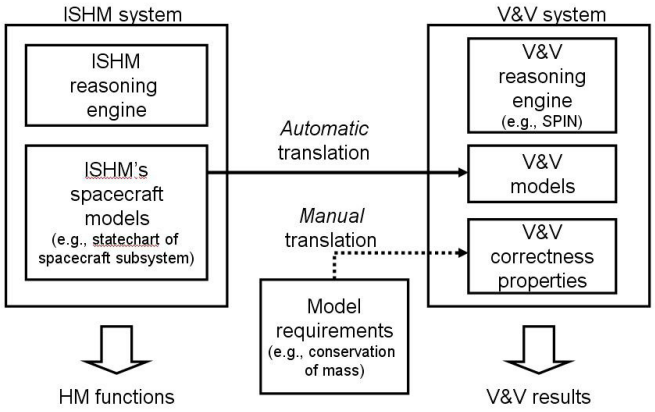


Figure 3: Translating ISHM Models for V&V

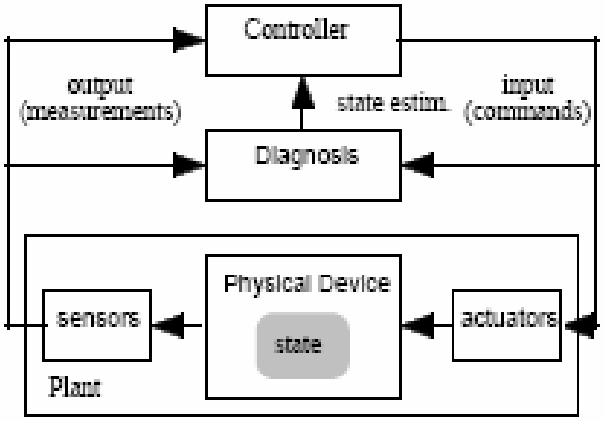


Figure 4: Architecture of a Diagnosis System
(from [Cimatti et al, 2003])

Traditional techniques, such as testing, can also leverage the availability of such models. For example, [20] describes test automation (generation of the test cases, test drivers, and test result evaluation) utilizing models, demonstrated on the ill-fated Mars Polar Lander software design. Human-conducted activities such as reviews and inspections may be well-suited to scrutiny of declarative models.

Another source of opportunity offered by model-based reasoning is that the reasoning software can yield both its result (e.g., a diagnosis), and the chain of reasoning that led to that result. That chain of reasoning provides opportunities for cross-checking – not only checking that the result is correct, but also that it is correct for the right reasons (e.g., all the appropriate information was taken into account when arriving at its conclusion). For an example of this used during testing of a spacecraft experiment's AI planner, see [21].

Diagnosability—An important property of ISHM systems is that they are adequate to support diagnosis of a specified class of faults. Often termed *diagnosability*, this means that using the information available from sensors, the ISHM system is always able to distinguish whenever the system is in a fault state, and if so, disambiguate which fault state it is. Note that this is a property of a combination of the spacecraft system (what states it can exhibit), the sensors (what information about the system state they make available to ISHM), and the reasoning capabilities of the ISHM system. For example, if among the spacecraft system's possible behaviors there are two scenarios that lead to system states required to be distinguished, and yet the sensor information made available to the ISHM system is exactly the same for both those scenarios, then it would clearly be impossible for the ISHM system to make the distinction. For a discussion of diagnosability and approaches to its attainment, see [22] and [23]. An approach to verification of this property is described in [24].

For V&V of the system as a whole, [25] and [26] discuss an approach that focuses on advanced simulation of the actual software (as opposed to verification of the model only). Concretely, this has been implemented in the Livingstone PathFinder (and Titan PathFinder) framework. Although this approach does not address diagnosability directly, it can catch diagnosis errors that may be traced back to diagnosability issues. They discuss an application of this approach to the main propulsion feed subsystem of the X-34 space vehicle.

Model Checking—Earlier in this section we discussed applications of model checking to ISHM. Here we take a closer look at this technology.

Model checking [27] is a method for formally verifying finite state systems. Model checking can be applied to designs for both hardware and software, and, more recently, to application source code. Historically, model checking was initially applied to communication protocol verification. Since then, particularly since the Intel Pentium bug, model checking has become a standard industrial verification technology for hardware, with hundreds of practitioners.

In this paper, we focus on the application of model checking to software, a field that has developed rapidly over the past eight years. Model checking is especially aimed at the verification of reactive, embedded systems, that is, systems that are in constant interaction with their environments. ISHM is an example of such a system. These systems are extremely hard to test using traditional techniques, which usually require hand-generated test cases.

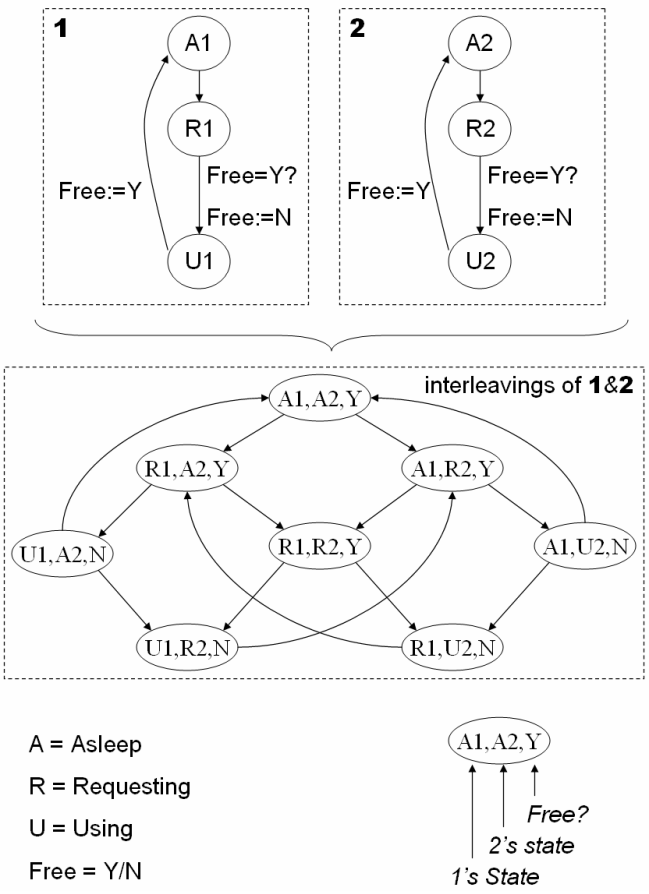


Figure 5: State space of an example concurrent system with two activities

One reason these systems are hard to test is the nondeterminism resulting from multithreading. In contrast to testing, model checking can verify such systems because it can explore the relevant part of the system's state space, including nondeterministic branching.

Figure 3, shows two very simple activities, each of which from time to time requests use of the same resource (this illustration is based on an example presented in [28]). Each activity transitions (boxes 1 and 2) between three states: "A" for Asleep (the activity isn't doing anything), "R" for Requesting (the activity is preparing to use the shared resource), and "U" for Using (the activity is using the shared resource). A shared variable "Free", which can take on values "Y" or "N", is used as a semaphore by these two activities in such a way that they never are both using the shared resource at the same time.

Each activity can be in one of three states, and the variable can hold one of two values, so in principle the state-space of their combination is 3x3x2 = 18 states. However, from the starting point of both activities asleep and Free=Y, not all of these 18 states are reachable (essentially because either activity's transition from Requesting to Using can take place only if Free=Y). The lower box "interleaving of 1&2" shows the eight states that are reachable from one another.

Model checking takes as input the activities themselves, and is able to completely explore the reachable state space. By so doing, it can answer questions such as "is it *guaranteed* that activities 1 and 2 are never both in the Using state at the same time?", and "is it *always possible* to get back to the initial state (A1,A2,Y)?". From visual scrutiny of the interleavings' state-space it is obvious that the answer to both of these is "yes". However, in realistic problems the state-space is *vastly* larger, so drawing this out by hand is infeasible, and traditional testing wouldn't cover more than a small fraction of possible paths. Model checking nevertheless works in such situations by a combination of:

- Very efficient representation of states and transition graphs (in some forms of model checking whole sets of states are manipulated at a time), utilizing advances from the computer science field (hash tables, binary decision diagrams, ...),

- Avoiding the need to explore all paths when the model checking algorithm can ascertain that the result down one path would be the same as the result down an already explored path, and

- Abstraction – removal or approximation of "irrelevant" portions of the state space and/or the data within the states (e.g., substitution of integer values with simply negative/zero/positive). Abstraction is key to scaling model checking to larger problems, and is an area of active research.

Although originally developed for finite state systems, model checking has been extended to work with at least some infinite state systems and also with real-time systems. Model checking can verify simple properties such as reachability and freedom from deadlock (is deadlock avoided in the system), as well as more complex properties such as safety (the system never gets into a state with an undesired property such as exceeding a resource limit) or liveness (for example, every request eventually obtains a response).

A key step in model checking is to describe the system in a state-based, formal way. Typically, a model checker will require a special purpose system description language be used for this purpose. More recently, ways have been developed that allow model checking to work from programming languages such as Java, C, and C++, thus enabling model checking to be applied directly to source code.

Another approach to easing the task of developing the state-based model is to translate from an alternative state-based modeling representation already in use. Pioneering work in

this direction was to use StateMate® Statecharts as the starting point, and translate from them into the input language of a model checker [29] and [30]; [31] did similar work starting from UML's statechart diagrams. Translation to model checking notations is also evident in other state-based notations in use in the formal methods community, e.g., SCR [32], and RSML [33].

Generally the system description is at a higher level of abstraction than the eventual source code. Thus, some steps must be taken to ensure the description is consistent with the code.

Once the system has been modeled in this way, model checkers can usually verify certain common properties such as reachability of all states in the model, freedom from deadlock and lack of race conditions. If more complex properties are to be verified, the next step is to express them as specifications using some form of temporal logic. This logic deals with properties of states and execution paths. For example, you can assert that a property is true in some future state or in all future states.

Once the system is modeled and the temporal logic specifications are written, the model checker is applied. It traverses the state space of the model and determines whether the specifications hold. Very large state-spaces can often be traversed in minutes. The technique has been applied to complex industrial systems, ranging from hardware to communication protocols to safety-critical plants and procedures.

Tools for model checking—Important examples of model checkers are SPIN, NuSMV, VeriSoft, Java PathFinder and UPPAAL.

Spin [34] is by far the most popular software model checker. It was designed to test the specifications of concurrent, distributed systems—specifically communications protocols [35], though it is applicable to any concurrent system. It detects deadlocks, busy cycles, conditions that violate assertions, race conditions, and unwarranted assumptions about the relative speeds of processes. Spin was developed at Bell Labs in the formal methods and verification group starting in 1980. Spin targets efficient software verification, not hardware verification. It uses a high level language to specify systems descriptions (PROMELA - PROcess MEta LAnguage). PROMELA is the closest model-checking specification language to a real programming language. Spin can be used as a full linear temporal logic (LTL) model checking system, supporting all correctness requirements expressible in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL.

Spin has been used to trace logic design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, and railway signaling protocols. Spin uses an "on-the-fly" approach where not all of the model needs to be in memory at once. The tool implements techniques to allow it to scale to handle real applications.

An important limitation on the use of Spin with existing applications is the need to translate the applications to PROMELA. Several NASA applications have been translated to PROMELA and analyzed using Spin. NASA experience reports can be found in [36] and [37].

Spin developers have implemented partial automatic translation to address this problem. The FeaVer tools [38] support translation of C to PROMELA, and are oriented toward applications in the maintenance phase of the lifecycle. Spin and related tools and documentation are available at [39].

Another model checker, NuSMV (New Symbolic Model Verifier)[40], [41], is a symbolic model checker developed as a joint project among the Center for Scientific and Technological Research in Trent, Italy, Carnegie-Mellon University, the University of Genoa and the University of Trent. The NuSMV system requires specifications to be written in computational tree logic (CTL). The input language of NuSMV is designed to allow the description of finite state systems that range from completely synchronous to completely asynchronous, and from the detailed to the abstract. The language provides for modular hierarchical descriptions, and for the definition of reusable components. CTL allows safety, liveness, fairness, and deadlock freedom to be specified syntactically.

Most model checkers have not been specialized for handling real-time systems, where timing aspects are critical. UPPAAL[42] is a model checker that supports real-time constraints and has been applied to verification of plan models [43], real-time controllers, and communication protocols [44].

Another approach to model checking is to apply the technique to the program itself. Java PathFinder (JPF) [45] is a model checker that takes Java byte code as its model. JPF has been under research and development for seven years in the Robust Software Engineering group at NASA Ames Research Center and has been applied to a number of NASA applications. These include not only Java but also Lisp and C++ applications that were hand-translated into Java. In each of these, it has found defects such as a time-partitioning error (in the DEOS operating system) [37]; and most of the known, and some previously-unknown, concurrency errors in the Mars K9 Rover code. JPF can handle the full Java language, but is subject to engineering limitations such as the size of programs it can handle. In mid-2005, JPF was released under an Open Source license

[46]. JPF is also being extended to handle C++ and has recently been applied to a C++ flight software application for the Space Shuttle. As with the other major model checking technologies, there is an active research community developing JPF extensions, such as abstraction tools and heuristics for state space search.

Other major research model checkers oriented toward programming languages include:

- SLAM (for C) [47]
- Verisoft (for C) [48]
- BLAST (for C) [49]
- dSPIN (Spin-based model checker for a subset of C++) [50]
- Bandera (analysis and abstraction tools for Java programs that derive models that can serve as input to variety of model checkers) [51].

Planning and Plan Execution Systems in ISHM

In addition to diagnosing the health status of the systems they monitor, many ISHM systems will be required to plan the appropriate actions to recover from unhealthy states, and to execute those actions. Model-based techniques will play an increasingly prominent role in the planning and execution stages, just as in the diagnosis [19]. Artificial Intelligence techniques for response planning have the same reasoning engine + models architecture, and so are prone to the same V&V challenges and opportunities as diagnosis systems. In addition, a plan execution system ("executive") is needed to execute the plans. V&V of this software system must ensure that the execution of the commands and the response of the fault protection system conform to pre-planned behavior. [52] discusses an executive built with plan verifiability in mind. [53] describes the results of applying several verification tools to an executive for a robotic Martian rover.

Core Algorithms

In addition to the verifying the AI components, V&V of ISHM will require assuring the correctness of its core algorithms (e.g., fault-tolerance, voting schemes); this kind of problem has long been appropriate for formal methods such as theorem proving (e.g., [54]; [55]; [56]).

General-purpose Program Verification Techniques

A large number of general-purpose program verification techniques commonly applied to many forms of software are equally applicable to ISHM software. We do not attempt to summarize them in this paper. Here we list several general-purpose techniques that are seeing increased use, and that we believe are well-suited to aid in the verification of ISHM's procedural code.

Static Analysis—Static analysis detects defects by analyzing an application's source code without actually executing it (e.g., by tracing through the program's source-code statements to ascertain whether every use of a variable is preceded by its initialization). This means that static analysis can make determinations that apply to all possible executions (in contrast to testing, which typically explores only a subset of possible executions).

Static analysis tools have been available almost as long as compilers; the earliest statically-detected defects were syntax errors reported by compilers. The capabilities of static analysis have grown significantly, and there are now static analyzers that can detect a wide range of programming errors such as use of uninitialized variables, potential division by zero, memory leaks, and pointer mismanagement. A major area of research in static analysis is *abstract interpretation* [57]. This research has spawned tools, both research prototypes and commercial products, that attempt to enable more precise defect detection. A summary of currently available static analysis tools for C (including one based on abstract interpretation) can be found at [58].

Typically static analyzers detect only the most general classes of defects, that is, operations that are incorrect in any application. Nevertheless, such defects are common enough, and have sufficient potential to lead to failures, that reliable detection is important, particularly in aerospace applications—the well-known Arianne 501 loss was caused by an arithmetic overflow.

One drawback of static analyzers is the typically high rate of false positives—non-defects that are reported as defects—and indeterminate items that the static analyzer cannot definitively assert to be defects. These must be filtered one-by-one by the developers themselves, a burdensome task. An emerging approach to reducing the frequency of such false positives and indeterminate items is to combine user feedback, machine learning and statistical analysis to reduce the error rate in static analysis [59]. A commercial product is available that incorporates some of these methods [60].

Efforts have been made to extend the capability of static analysis by providing information to the analyzer beyond what is available, or readily derivable, from the source code, for example, legal ranges of values for input variables. Given this additional information, static analyzers can both reduce the number of false positives, and expand the classes of errors it can detect. Usually this additional information must be provided by the programmer as annotations in the source code (usually in the form of inline comments that are understandable to the analyzer). The Java Modeling Language (JML) is an example of a language for writing specifications for Java programs. The aim is to provide a

specification language that is easy to use for Java programmers and that is supported by a wide range of tools for specification type-checking, runtime debugging, static analysis and verification. [61] is an overview of JML tools and applications. A static analyzer built around JML is the Extended Static Checker for Java (ESC/Java) [62]. ESC/Java 2 [63] attempts to integrate ESC-style static analysis with interactive verification to prove that a program is correct with respect to its JML specification.

Runtime Analysis—Methods that can expand the information gained from individual test cases would be useful for testing of the numerous behaviors that ISHM systems can exhibit – an example of such a method is the recognition of inconsistent uses of shared variables in a test run, even if no classical race condition occurs within that run [64], [65]. [66] describes a similar approach to deadlock detection.

Testing

We have argued that traditional V&V methods, with their heavy reliance on testing, will not scale well to the nature of advanced ISHM. Nevertheless, testing will remain an indispensable element in the V&V arsenal, and means exist to amplify its efficacy. This was seen in the V&V of NASA's "Remote Agent" – a spaceflight demonstration of AI technology used to automatically control a spacecraft. The V&V team's approach to testing is described in [67]. The two areas their approach (and their experience applying it) emphasized were of careful selection of test cases, and judicious use of testbeds (with varying levels of fidelity, availability and speed). In their selection of test cases they applied the following techniques:

- Informal, but expert reasoning about the places where a test case could be taken as representative of a neighborhood of similar tests (thus obviating the need to run those other tests),
- Downsizing the test space based on reasoning that rules out cases not relevant to the mission plan (for example, discarding those that start from what would not be possible starting conditions in the actual mission profile),
- Down-selection of a set of tests that, as a whole, exercise all combinations of nominal, single variations from nominal, and some (manually chosen as key) instances of multiple simultaneous variations from nominal, test cases
- Metrics to double-check plausible notions of "coverage" that a test suite provides, and
- Automation to speed up test running and checking of the results of testing.

Software Reliability Engineering

ISHM systems may be expected to be amenable to traditional software reliability engineering techniques based on measurements of defect discovery and removal during development and test: see [68], [69] for overviews of this field.

It is plausible to expect that ISHM systems' models themselves may also be amenable to measurement-based techniques, for example, by tracking the number of corrections made to the models as they undergo inspection, review, simulation, analysis, and, finally, part of the system testing process.

4. DEVELOPMENT APPROACHES

Many development factors have a direct effect on how readily the software product can be V&V'd. For example, a poor choice of development platform (such as a legacy programming language) may render many V&V technologies inapplicable.

Since V&V typically accounts for 50 - 75% or more of the software lifecycle costs [70], development decisions should take into account the effect on these costs, even when initial lifecycle stage costs may be higher as a result. This is particularly true for long-lived components that can be expected to undergo maintenance for many years or even decades.

Traceability of Requirements

Standard development practices call for requirements and designs to be captured and scrutinized (e.g., via inspections and reviews), and for traceability among them to be maintained. Traceability often takes the form of a matrix, for example, a "requirements traceability matrix" (between requirements, and system functionality), and a "requirements test matrix" (between requirements and test cases). Use of the latter helps promote the expression of requirements in a testable form, and helps ensure that the test plan covers all the requirements.

Beyond these standard practices, the software engineering community is pursuing more formal (formal in the sense of "mathematical", not formal in the sense of "formal inspections") treatments of requirements traceability. [71] provides an overview of this field. These methods help by encouraging a more thorough elicitation of relevant information, and allowing the application of machine-supported mechanisms to check the correctness of requirements decomposition (the traceability between requirements at one level of expression, and their equivalent at the next, more detailed, level). The relevance of this is highlighted by NASA's Mars Polar Lander, thought to have failed because of a problem that traces back to incorrect mapping of system requirements to flight software requirements ([72], page 120).

V&V of Design Models

Model-based design is an area of growing attention within both the software engineering community, and the broader systems engineering community. In the software engineering milieu, several early proponents of various modeling notations worked together to achieve a consensus in the form of the Unified Modeling Language (UML) [73], around which a great deal of work has since coalesced. UML's "models" are expressed via diagrams, the complete set of which encompass the expression of structure, behavior and interaction of the modeled system. Of most relevance to ISHM's V&V challenges, UML's *behavioral* models take the form of sequence diagrams, statechart diagrams (a variant of the Statechart concept originally propounded in [74]) and activity diagrams. Together, these provide a set of notations with a commonly agreed upon semantics. Multiple tool vendors now offer support for UML (e.g., development environments such as Rhapsody, Tau, etc.)

In the systems engineering milieu, there an analogous trend towards more machine-manipulable representations of development artifacts and their interrelationships. For example, SysML [75] is a visual modeling language, in a similar spirit to UML, but designed for systems engineering applications. Other examples of tool-supported systems engineering notations include CORE®, from Vitech Corporation [76], and Cradle®, from 3SL [77].

The key point is that the adoption of model based design techniques is leading to the capture of much more of the design information in machine-manipulable formats. Analysis techniques that help V&V can take advantage of the availability of this information. In an earlier section we discussed how the analysis method of "model checking" can take advantage of existing uses of statechart notations as source for the formal state-model input it requires. The other notations of systems modeling are also amenable to analysis, notably checking consistency and completeness among the information captured in the multiple diagrams that document a system design [78], provided those diagrams are ascribed a sufficiently formal (mathematical) interpretation [79]. A further advantage they offer is early feedback through simulation of the models themselves, well in advance of the creation of production quality code. Typically, such feedback takes the form of scenarios (possible execution paths through the model); when scrutinized by domain experts, the concrete nature of such scenarios makes it is easy for the experts to recognize mistakes (e.g., "it shouldn't have done that at that point..."). If coupled with some form of animation that renders visible the execution's effects and actions, this becomes particularly effective and palatable [80], [81]. Just as there are many ways to exercise a piece of software during testing, there are many ways to exercise a model during simulation. Deciding which simulations to perform is an area of ongoing research: [82], [20]; [83] combines this with runtime monitoring, and describes an application to a NASA rover controller; [84] apply the approach to a fault protection system used on several NASA spacecraft.

Prototyping Environment

Prototypes are often constructed for flight software prior to the deployed implementation. The development environment for prototypes need not be the same as the one selected for building the deployed implementation. In fact, re-use of the prototype as the production system is to be discouraged if the prototype is to be used as a test oracle ("a mechanism for determining behavioral correctness of test executions" [85])—the test oracle and the deployed implementation should not share the same defects.

Often the selection of the development environment for the deployed implementation is driven by the lack of choice in deployment (runtime) platforms, the need to integrate with other flight applications and hardware, the need for highly optimized code, the infrastructure costs to switch to a new development environment, and risk aversion toward changing what has worked in the past.

On the other hand, prototypes are developed with an emphasis on speed-of-development to demonstrate correct algorithms, show the requirements can be met, and help resolve issues with the requirements. For these purposes, the development environment for prototyping should support rapid development and a wide range of V&V tools, and the ability to quickly implement user interfaces that assist in human analysis, but that may not be needed in the target environment.

The prototype should be portable among different platforms and its development environment should be vendor-independent. To limit the V&V effort (as well as implementation effort), development should avoid implementing functionality that is already available in libraries (such as container classes, IO libraries, and user interface toolkits), and should avoid implementing functionality that is available in the runtime environment, including memory management and reflection.

Coding Standards

Regardless of the languages selected for ISHM implementation, observance of coding standards is critical for enabling V&V. For example, avoiding deeply nested pointers makes static analysis easier. In object-oriented languages, multiple inheritance complicates program analysis. General (language-oriented) coding standards are well-known (for example, [86]). However, for mission- and safety-critical code, where there is a heavier burden on V&V, the "standard" coding standards should be augmented to encourage design and coding practices that are consistent with the requirements of V&V technologies. Fortunately there are "shallow" static analyzers that can detect most violations of coding standards [87] and some of these can

be extended to incorporate new standards. An extensive discussion of the risks in using the C++ language for safety critical systems, as well as mitigations for some of these risks, is provided in [88] and [89], which focus on defining a "safer" subset of C++ together with the use of coding standards, static analysis, testing tools, and manual design and code reviews.

Aspect-Oriented Programming

Our experience with object-oriented flight software for spacecraft is that characterizing and observing valid software behavior during testing can be very difficult. This is particularly true for software that performs situational awareness, where the limited outputs conceal a very large internal state space. In addition, the programming language features that promote good design by hiding the internal state of objects from other objects in the system can limit what the test harness objects can see. An obvious solution is to integrate the test code into the objects so there is greater visibility. However, the result is that test code is spread around throughout the system and is more difficult to maintain and remove.

Aspect-Oriented Programming (AOP) [90], [91] can be applied to obtain greater visibility into the internal state while at the same time avoiding proliferation of test code in the program's source files[92]. Once the state properties of interest are identified, the software can be instrumented using AOP tools to monitor these properties at specified points in the program. By maintaining the instrumentation as aspects, it is possible to separately maintain the test code and easily insert it into new releases. The aspect code resides in its own set of source code files, and it is not necessary to modify the target application to support testing. This allows easily switching between the instrumented and uninstrumented code, or selecting arbitrary groups of instrumentations to be inserted. Again without proliferating test code in the target application, AOP tools provide visibility into the state of objects in the system under test.

5. CONCLUSIONS

ISHM software for spacecraft faces significant V&V challenges for several reasons: most of the scenarios that ISHM is designed to manage are *off-nominal*; and there is a large number of off-nominal scenarios. Human rating requirements state that testing is required to "verify and validate the performance, security, and reliability of all critical software across the entire performance envelope", which includes much of the ISHM software's state space. In addition, ISHM must not only detect and handle off-nominal scenarios, but it must reliably avoid "false alarms", especially when the response to a false alarm may cause loss of mission or crew. In addition to these externally-imposed requirements, the nature of ISHM software architecture, design and implementation poses its own challenges to V&V. Traditional V&V practices generally do not scale up to handling these challenges.

However, there are emerging V&V technologies that may address most aspects of these challenges. These include model-based reasoning to V&V ISHM models; plan execution systems developed with verification in mind; formal methods for verifying key algorithms; and general purpose program verification techniques.

Regardless of the V&V techniques to be used, early development decisions can defeat effective V&V, and thus it is essential to plan for incorporating maturing V&V technologies and development technologies that have an impact on V&V costs. Even if specific V&V technologies are not yet certified, development should incorporate practices such as capturing requirements and designs in a machine-manipulable manner; ensuring traceability from software artifacts to requirements; and selecting prototyping environment that provides extensive support for emerging V&V technologies.

6. ACKNOWLEDGEMENTS

The research described in this paper was carried out at NASA Ames Research Center and at the Jet Propulsion Laboratory, California Institute of Technology, and was funded by both the National Aeronautics and Space Administration and by the Jet Propulsion Laboratory's internal Research and Technology Development program. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

7. REFERENCES

[1] Griffin, M. Remarks at 2005 International Astronautical Conference, Fukuoka, Japan, Oct. 17, 2005. Reported at http://www.spaceref.com/news/viewpr.html?pid=18071

[2] Northrop Grumman Corporation. "Northrop Grumman-Boeing Team Unveils Plans for Crew Exploration Vehi-cle", SpaceRef.com, Oct. 12, 2005. http://www.spaceref.com/news/viewpr.html?pid=18022

[3] Committee to Identify Potential Breakthrough Technologies and Assess Long-Term R&D Goals in Aeronautics and Space Transportation Technology, National Research Council *Maintaining U.S. Leadership in Aeronautics: Breakthrough Technologies to Meet Future Air and Space Transportation Needs and*

Goals. National Academy Press, Washington, D.C., 1998.

[4] Hayhurst, K.J. & Holloway, C.M. "Challenges in Software Aspects of Aerospace Systems," in *Proc. 26th Annual IEEE NASA Goddard Software Engineering Workshop*, pp. 7-13, Nov. 2001.

[5] Feather, M., Fesq, L., Ingham, M., Klein, S., Nelson, S. "Planning for V&V of the Mars Science Laboratory Rover Software," in *Proc. IEEE Aerospace Conference*, 2004.

[6] Markosian, L., Feather, M., Brinza, D. "V&V of ISHM for Space Exploration," presented at *First Int'l Forum on Integrated System Health Engineering and Management in Aerospace*, 2005.

[7] Martin, G. & Shukla, S. "Panel: hierarchical and incremental verification for system level design: challenges and accomplishments," in *Proc. MEMOCODE '03, Formal methods and models for Co-Design*, pp. 97-99, 2003.

[8] de Boer, F. & Bonsangue, M. (eds). *Proc. Workshop on the Verification of UML Models*, Oct 2003, *Electronic Notes in Theoretical Computer Science*, Volume 101, pp 1-179, 2004.

[9] Giannakopoulou, D. & Penix, J. "Component Verification and Certification in NASA Missions," in *Proc. 4th ICSE Workshop on Component-Based Software Engineering*, 2004.

[10] Cobleigh, J., Giannakopoulou, D. & Pasareanu, C. "Assume-guarantee Verification of Source Code with Design-Level Assumptions," *Proc. ICSE'04: 26th Int'l Conf. on Software Engineering,* Edinburgh, Scotland, pp. 211-220, May 23-28, 2004.

[11] Lindsey, T. & Pecheur, C. "Simulation-Based Verification of Autonomous Controllers with Livingstone PathFinder," in *Proc. TACAS'04: Tenth Int'l Conf. on Tools And Algorithms For The Construction And Analysis Of Systems, Springer LNCS*, vol. 2988, Barcelona, Spain, pp. 357-371, 2004. http://ti.arc.nasa.gov/ase/papers/TACAS04/lpf-tacas04.pdf

[12] Feather, M.S., Fickas S. & Razermera-Mamy, N-A. "Model-Checking for Validation of a Fault Protection System," in *Proc. IEEE 6th International Symposium on High Assurance Systems Engineering*; Boca Raton, Florida, Oct. 2001, pp. 32-41.

[13] Menzies, T. & Pecheur, C. "Verification and Validation and Artificial Intelligence," in Zelkowitz, M. (ed.), *Advances in Computers*, Volume 65. Elsevier, 2005.

[14] John Rushby. *Formal Methods and the Certi_cation of Critical Systems*. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993. Also issued under the title "Formal Methods and Digital Systems Validation for Airborne Systems" as NASA Contractor Report 4551, December 1993. URL http://www.csl.sri.com/papers/csl-93-7/

[15] Schneider, F., Easterbrook, S.M., Callahan, J.R. & Holzmann, G.J. "Validating requirements for fault tolerant systems using model checking," in *Proc. 3rd Int. Conf. on Requirements Engineering*, 6-10 Apr 1998, pp 4-13. http://gltrs.grc.nasa.gov/cgi-bin/GLTRS/browse.pl?1990/CR-185259.html

[16] Pingree, P. J., Mikk, E., Holzmann, G. J., Smith, M. H. & Dams, D. "Validation of Mission Critical Software Design and Implementation using Model Checking," in *Proc. 21st Digital Avionics Systems Conference*, Volume 1, pp 6A4-1 – 6A4-12, Oct 2002.

[17] Pecheur, C. & Simmons, R. "From Livingstone to SMV: Formal Verification for Autonomous Spacecrafts," in *Proc. 1st Goddard Workshop on Formal Approaches to Agent-Based Systems*, pp 5-7, April 2000.

[18] Williams, B. C. & Nayak, P. P., "A Model-based Approach to Reactive Self-Configuring Systems," in *Proc. AAAI-96*, 1996.

[19] Penix, J., Pecheur, C., & Havelund, K. "Using Model Checking to Validate AI Planner Domain Models," in *Proc. SEL'98: 23rd Annual Software Engineering Workshop,* NASA Goddard, Dec. 1998. http://ti.arc.nasa.gov/ase/papers/SEL98/Penix-SEL-Workshop.pdf

[20] Blackburn, M., Busser, R., Nauman, A., Knickerbocker, R. & Kasuda, R. "Mars Polar Lander fault identification using model-based testing," in *Proc. 8th IEEE International Conference on Engineering of Complex* Computer Systems, 2-4 Dec. 2002 pp. 163-169.

[21] Feather, M.S. & Smith, B. "Automatic Generation of Test Oracles – From Pilot Studies to Application," in *Proc. Automated Software Engineering* (Kluwer); Vol 8, No. 1, Jan 2001, 31-61.

[22] Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., & Teneketzis, D.C. "Diagnosability of

discrete-event systems," *IEEE Transactions on Automatic Control, vol.* 40, no. 9, pp. 1555-1575, 1995.

[23] Jiang, S. & Kumar, R. "Failure Diagnosis of Discrete Event Systems with Linear-time Temporal Logic Specifications," *IEEE Transactions on Automatic Control*, vol. 49, no. 6, June, 2004.

[24] Cimatti, A., Pecheur, C. & Cavada, R. "Formal Verification of Diagnosability via Symbolic Model Checking," in *Proc. IJCAI 2003: 18th Int'l Joint Conf. on Artificial Intelligence,* Acapulco, Mexico, pp. 501-503, Aug. 9-15, 2003 http://ti.arc.nasa.gov/ase/papers/MOCHART02/VVDiag-ECAI.pdf

[25] Lindsey, T. & Pecheur, C. "Simulation-Based Verification of Livingstone Applications," in *Proc. DSN 2003: Int'l Conf. on Dependable Systems and Networks*, San Francisco, CA, pp. 741-750, June 22-25, 2003. http://ti.arc.nasa.gov/ase/papers/DSN03/lpf-dsn03.pdf

[26] Lindsey, T. & Pecheur, C. "Simulation-Based Verification of Autonomous Controllers with Livingstone PathFinder," in *Proc. TACAS'04: Tenth Int'l Conf. on Tools And Algorithms For The Construction And Analysis Of Systems, Springer LNCS*, vol. 2988, Barcelona, Spain, pp. 357-371, 2004. http://ti.arc.nasa.gov/ase/papers/TACAS04/lpf-tacas04.pdf

[27] Clarke, E.M., Grumberg, O. & Peled, D. *Model Checking.* MIT Press, 1999.

[28] Visser, W. "Sofware Model Checking", available at http://ase.arc.nasa.gov/visser/ASE2002TutSoftwareMC-fonts.ppt

[29] N. Day. *A Model Checker for Statecharts (Linking CASE tools with Formal Methods)*. Technical Report 93-35, October 1993, Integrated System Design Laboratory, Dept. of Computer Science, University of British Columbia. http://citeseer.ist.psu.edu/cache/papers/cs/14916/ftp:zSzzSzftp.cs.ubc.cazSzftpzSzlocalzSztechreportszSz1993zSzTR-93-35.pdf/day93model.pdf

[30] E. Mikk, Y. Lakhnech, M. Siegel, G.J. Holzmann, "Implementing Statecharts in Promela/SPIN," in *Proc. Workshop on Industrial Strength Formal Techniques*, Boca Raton, Florida, Oct. 1998.

[31] Latella, D., Majzik, I. & Massink, M. "Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model checker," in *Formal Aspects of Computing, The International Journal of Formal Methods.* Springer. Vol. 11, no. 6, pp 637-664, 1999. http://www.inf.mit.bme.hu/FTSRG/Publications/fac99.pdf

[32] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer & R. Bharadwaj, "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications," *IEEE Transactions on Software Engineering* vol. 24 no. 11, pp 927-948, November 1998.

[33] B.J. Czerny & M.P.E. Heimdahl, "Automated Integrative Analysis of State-Based Requirements," in *Proc. 13th IEEE International Conference on Automated Software Engineering*, Honolulu, Hawaii, Oct. 1998.

[34] Holzmann, G. *The SPIN Model Checker*. Addison-Wesley, 2003.

[35] Holzmann, G. *Design and Validation of Computer Protocols*. Prentice-Hall, 1990. http://spinroot.com/spin/Doc/Book91.html

[36] Havelund, K., Lowry, M., Park, S., Pecheur, C., Penix, J., Visser, W. & White, J. "Formal Analysis of the Remote Agent - Before and After Flight," in *Proc. 5th NASA Langley Formal Methods Workshop*, Williamsburg, VA, 2000.

[37] Penix, J., Visser, W., Engstrom, E., Larson, A. & Weininger, N. "Verification of time partitioning in the DEOS scheduler kernel," in *Proc. 22nd Int'l Conference on Software Engineering,* 2000.

[38] Holzmann, G. & Smith, M. "A practical method for verifying Event-Driven Software," in *Proc. 21st Int'l Conference on Software Engineering,* 1999.

[39] http://www.spinroot.com

[40] Cimatti, A., Clarke, E., Giunchiglia, F., Pistore, M., & Roveri, M. "NuSMV: A New Symbolic Model Verifier". In *Lecture Notes in Computer Science*, no. 1633. Springer, Trent, Italy, 1999. http://nusmv.irst.itc.it/NuSMV/papers/cav99/html/index.html

[41] http://nusmv.irst.itc.it/NuSMV/

[42] Larsen, K., Larsson, F., Pettersson, P. & Yi, W. "Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction," in *Proc.18th IEEE Real-Time Systems Symposium*, 1997.

[43] Khatib, L., Muscettola, N. & Havelund, K. "Verification of plan models using UPPAAL," in *Formal*

Approaches to Agent-Based Systems, First Int'l Workshop, Greenbelt, MD, 2000.

[44] http://www.uppaal.com

[45] Brat, G., Havelund, K., Park, S. & Visser, W. "Java pathfinder—a second generation of a Java model checker," in *Workshop on Advances in Verification*, July 2000.

[46] http://javapathfinder.sourceforge.net

[47] http://research.microsoft.com/slam/main.htm

[48] http://www.bell-labs.com/project/verisoft/

[49] http://www-cad.eecs.berkeley.edu/~rupak/blast/

[50] http://www-verimag.imag.fr/~iosif/dspin/

[51] http://www.cis.ksu.edu/santos/bandera/

[52] Verma, V., Estlin, T., Jonsson, A., Pasareanu, C., & Simmons, R. "Plan Execution Interchange Language (PLEXIL) for Command Execution," in *Proc. International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS),* 2005. http://ti.arc.nasa.gov/people/pcorina/papers/vandi_isairas05.pdf

[53] Brat, G., Giannakopoulou, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C.S., Venet, A. & Wisser, W. "Experimental Evaluation of Verification and Validation Tools on Martian Rover Software," in *Proc. SEI Software Model Checking Workshop, Formal Methods in System Design*, vol. 25, issue 2, Pittsburgh, PA, pp. 167-198, Mar. 24, 2003

[54] Rushby, J. "Formal verification of algorithms for critical systems," in *Proc. Conference on Software for Critical Systems*, pp. 1-15, ACM, 1991.

[55] Pike, L. & Johnson, S. D. "The formal verification of a reintegration protocol," in *Proc. ACM Conference on Embedded Software (EMSOFT)*, September, 2005.

[56] Paul Miner, P., Geser, A., Pike, L. & Maddalon, J. "A unified fault-tolerance protocol," in *Proc. Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT),* LNCS 3253, Springer, 2004.

[57] Cousot, P. "Abstract Interpretation Based Formal Methods and Future Challenges," in *Informatics, 10 Years Back – 10 Years Ahead*. Wilhelm, R. (Ed.), Lecture Notes in Computer Science 2000, pp. 138 – 156. Springer, 2001.

[58] http://www.spinroot.com/static/

[59] Kremenek, T., Ashcraft, K., Yang, J. & Engler, D. "Correlation Exploitation in Error Ranking," in *Proc. 12th ACM SIGSOFT Twelfth Int'l Symposium on Foundations of Software Engineering*, Newport Beach, CA, 2004.

[60] "Fixing Software on the Assembly Line: An Overview of Coverity's Static Code Analysis Technology," Coverity, Inc., San Francisco, 2005. http://www.coverity.com

[61] Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G. T., Leino, K. M., & Poll, E. "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, Feb. 2005

[62] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., & Stata, R. "Extended static checking for Java," in *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002),* pp 234–245, 2002.

[63] Kiniry, J., Chalin, P., and Hurlin, C. "Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification," in *Proc. Verified Software: Theories, Tools and Experiments 05,* Zurich, 2005

[64] Artho, C., Havelund, K. & Biere, A. "High-level data races," in *Software Testing, Verification and Reliability* Vol 13, No 4, pp 207-227, Nov 2003.

[65] Artho, C., Havelund, K. & Bierre, A. "Using Block-Local Atomicity to Detect Stale-Value Concurrency Errors," *2nd International Symposium on Automated Technology for Verification and Analysis.* Taipei, Taiwan, October 2004.

[66] Bensalem, S. & Havelund, K. "Scalable Deadlock Analysis of Multi-Threaded Programs," *Parallel and Distributed Systems: Testing and Debugging 05*, Haifa, November 2005.

[67] Smith, B., Millar, W., Dunphy, J., Tung, Y., Nayak, P., Gamble, E., Clark, M. "Validation and Verification of the Remote Agent for Spacecraft Autonomy", in *Proc. IEEE Aerospace Conference (IAC 1999)*, Aspen, CO, March, 1999.

[68] *Software Reliability Engineering*, McGraw-Hill, New York, 1998.

[69] Vouk, M.A. "Software reliability engineering," in *Proc. 2000 Annual Reliability and Maintainability*

Symposium (RAMS), Los Angeles, CA, IEEE Computer Society.

[70] *The Economic Impacts of Inadequate Infrastructure for Software Testing.* National Institute of Technology and Standards, 2002.

[71] van Lamsweerde, A. "Goal-oriented requirements engineering: a guided tour," in *Proc. 5th IEEE International Symposium on Requirements Engineering*, pp. 249-262, Toronto, Ontario, 27-31 August, 2001.

[72] JPL Special Review Board. *Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions*. JPL D-18709, Jet Propulsion Laboratory, California Institute of Technology, March 2000.

[73] http://www.uml.org/

[74] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming* 8(3):231-274, 1987

[75] http://www.sysml.org

[76] http://www.vtcorp.com

[77] http://www.threesl.com

[78] C. Nentwich, C., Emmerich, W. , Finkelstein, A. & Ellmer, E. "Flexible Consistency Checking," *ACM Transactions on Software Engineering and Methodology*, vol. 12, pp. 28-63, 2003

[79] McUmber, W.E. & Cheng, B.H.C. "A General Framework for Formalizing UML with Formal Languages," in *Proc. 23rd International Conference on Software Engineering*, pp 433-442, May, 2001

[80] Brockmeyer, M., Jahanlan, F., Heitmeyer, C. & Winner, E. "A flexible, extensible simulation environment for testing real-time specifications," *IEEE Transactions on Computers*, 49(11), pp. 1184-1201, Nov 2000.

[81] Bachelder, E., & Leveson, N. "Animating Safety-Critical Automation Logic and Intent: a Candidate Design," in *Proc. 21st Digital Avionics Systems Conference*, pp. 7B1-1 – 7B1-10 vol 2, 2002

[82] Rayadurgam, S. & Heimdahl, M.P.E. "Coverage Based Test-Case Generation Using Model Checkers," in *Proc. 8th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pp 83-91, 2001.

[83] Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khursid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W. & Washington, R. "Combining test case generation and runtime verification," *Theoretical Computer Science,* vol. 336, pp. 209-234, 2005.

[84] A. Nikora & C. Heitmeyer, "Automated Specification-Based Test Case Generation Using SCR", Workshop on Software Engineering for High Assurance Systems, Portland, OR, May 2003 http://www.sei.cmu.edu/community/sehas-workshop/nikora2/

[85] Richardson, D.J., Aha, S.L. & O'Malley, T.O., "Specification-based Test Oracles for Reactive Systems," *Proc. 14th International Conference on Software Engineering*, Melbourne, Australia, pp 105-118, 1992.

[86] Meyers, S. *Effective C++, Third Edition. Addison Wesley Professional, 2005*.

[87] www.parasoft.com .

[88] Reinhardt, D. *Use of the C++ Programming Language in Safety Critical Systems*. MSc SCSE Project, University of York, UK, 2004.

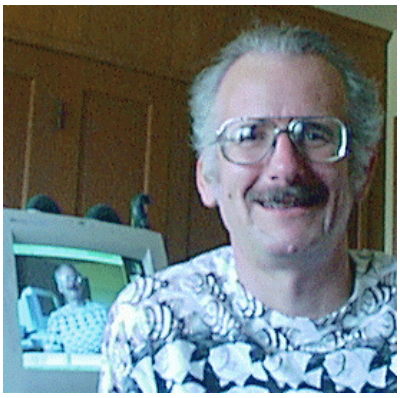
[89] Hill, M. & Whiting, E. *An investigation of the unpredictable features of the C++ language.* QINETIQ Ltd, Farnborough, UK, 2004

[90] Elrad, T., Filman, R. & Bader, A. "Aspect-Oriented Programming: Introduction," *Communications of the ACM,* 44(10), October, 2001.

[91] Filman, R., Elrad, T., Clarke, S. & Aksit, M. *Aspect-Oriented Software Development.* Addison Wesley Professional, 2004.

[92] O'Malley, O., Mansouri-Samani, M., Mehlitz, P. & Penix, J. "Seeing the Invisible: Embedded Tests in Code that Cannot be Modified," in *Proc. Info-Tech@Aerospace,* 2005.

BIOGRAPHY



Martin S. Feather *is a Principal in the Software Quality Assurance group at JPL. He works on developing research ideas and maturing them into practice, with particular interests in the areas of early phase requirements engineeering and risk management and of software validation (analysis, test automation, V&V techniques). He obtained his BA and MA degrees in mathematics and computer science from Cambridge University, England, and his PhD degree in artificial intelligence from the University of Edinburgh, Scotland. See http://eis.jpl.nasa.gov/~mfeather for further details.*

Lawrence Z. Markosian *is a Computer Scientist with QSS Group, Inc. at NASA Ames Research Center, where he led a team developing verification tools for C++. He is a member of the NASA Software Engineering Initiative's Research Infusion team. Prior to joining NASA, he was a founder of Reasoning Systems., where as VP of Applications Development he managed technology transfer of advanced software engineering tools. Markosian has an undergraduate degree in mathematics from Brown University and has done graduate work at Stanford University in logic and artificial intelligence.*